Electron Security

自定义协议

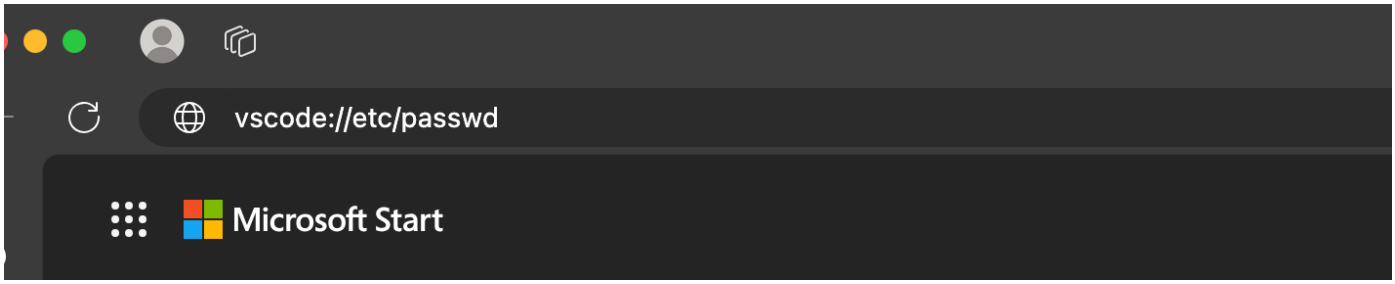
NOP Team

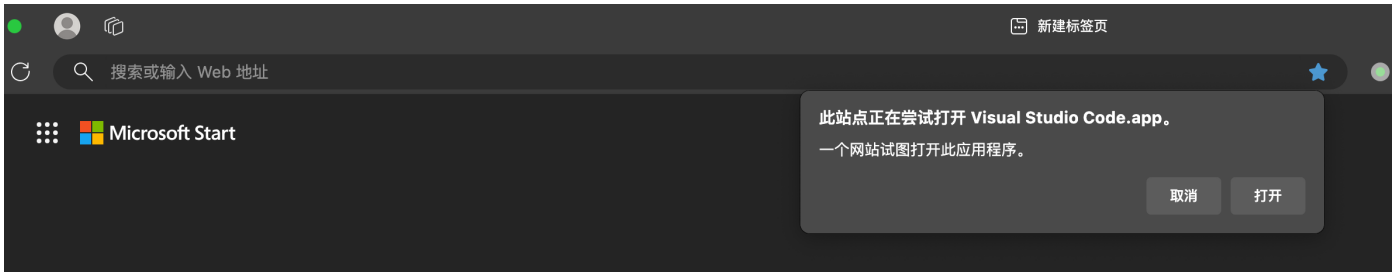
自定义协议 | ELectron 安全

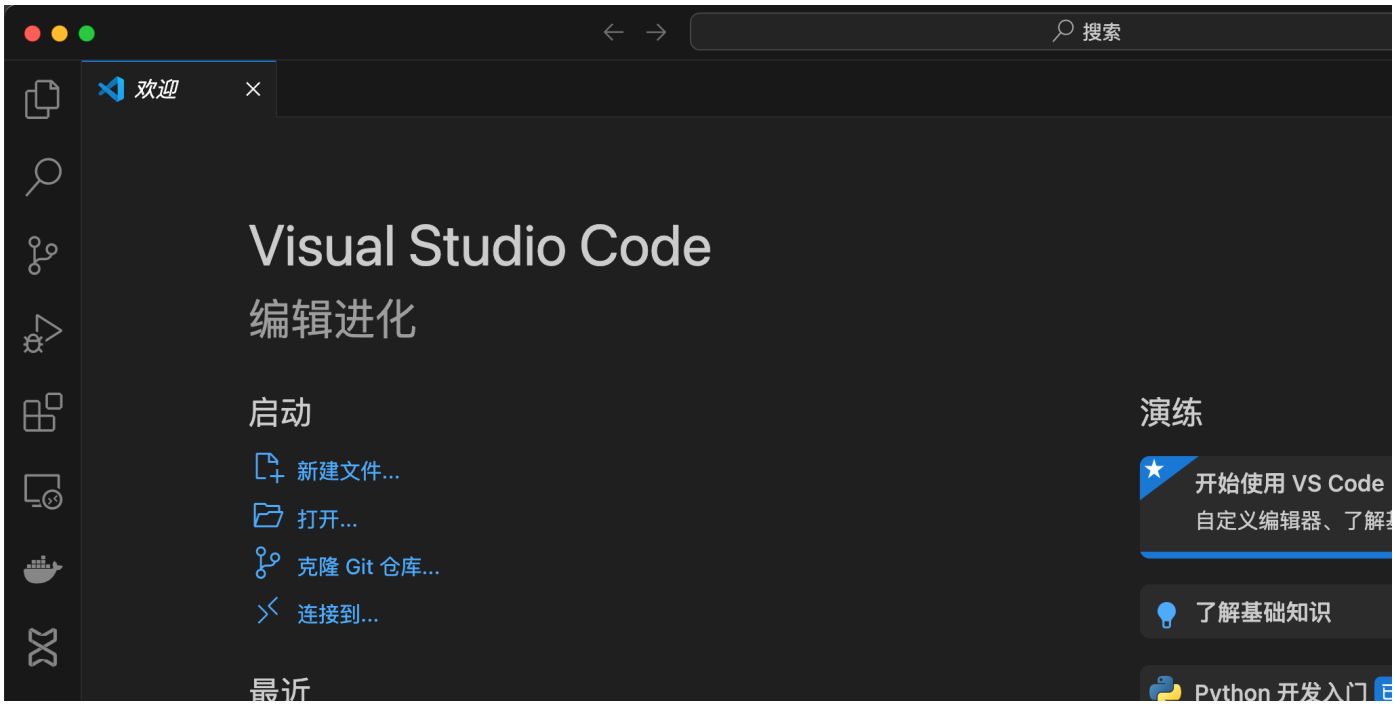
0x01 简介

大家好，今天和大家讨论的是自定义协议，在很多应用中，除了支持 `http(s)`、`file`、`ftp` 等开放的通用标准协议外，还会支持一些自定义协议，自定义协议常被用于实现特殊功能，比如深度集成应用程序与特定的网络服务、提升用户体验或实现安全的数据交换。

例如 `vscode` 就注册了 `vscode:` 协议，在浏览器中输入 `vscode://xxx` 就会唤醒 `vscode`







这就属于在系统层面全局注册了自定义的 `vscode:` 协议

在一些应用程序中，我们发现，调用资源不都是 `http(s)`、`file` 这种，尤其像是加载插件之类的操作，内部用的也是类似于 `vscode:` 这种协议，这种就属于应用内注册自定义协议

今天的内容也是围绕着这两种情况进行讨论

0x02 程序内部注册自定义协议

1. 效果展示

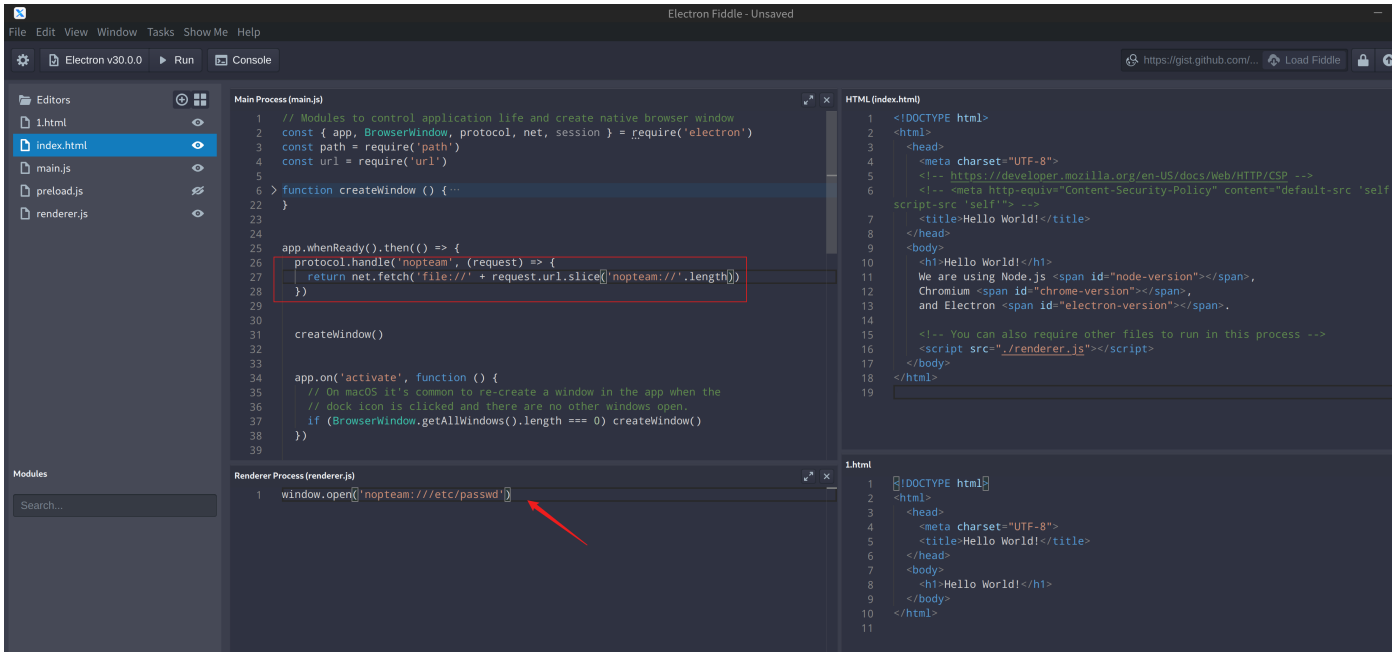
官方给了一个案例，让我们可以注册一个和 `file` 协议相同效果的协议

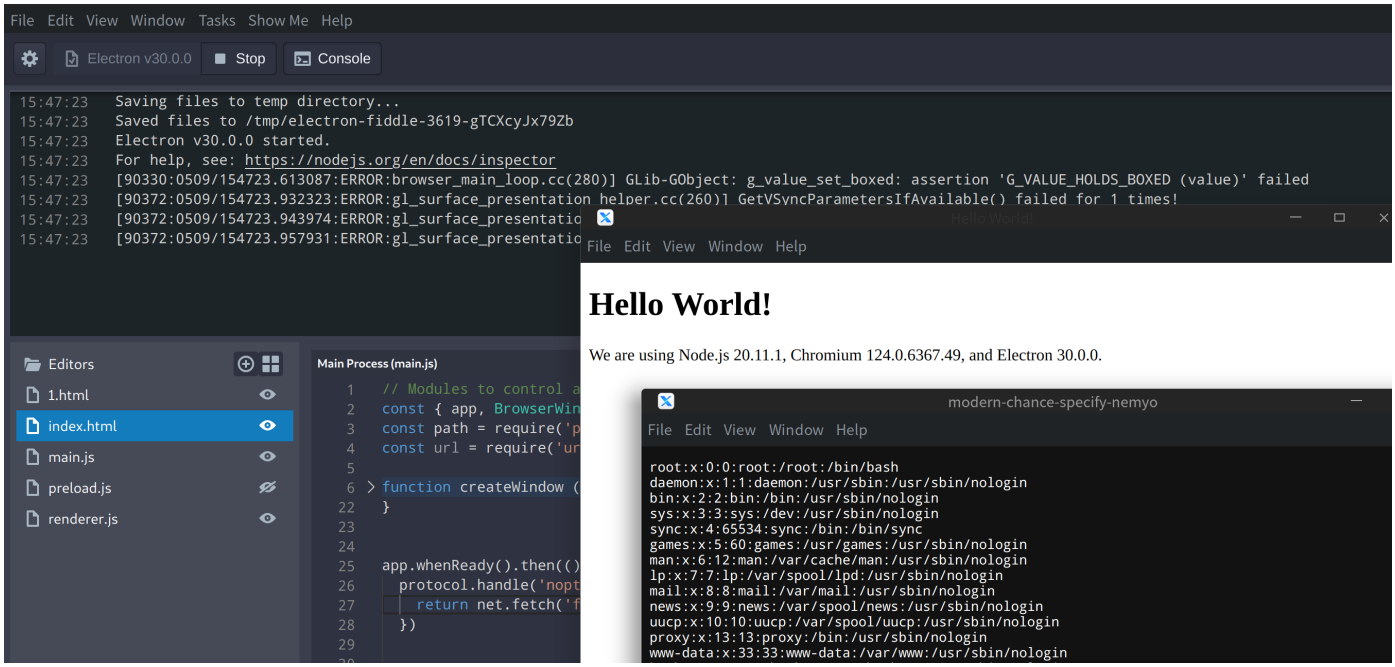
```
const { app, protocol, net } = require('electron')

app.whenReady().then(() => {
  protocol.handle('atom', (request) =>
    net.fetch('file://' + request.url.slice('atom://'.length)))
})
```

这里是注册了一个 `atom` 协议，我们修改为 `nopteam` 协议，嘿嘿

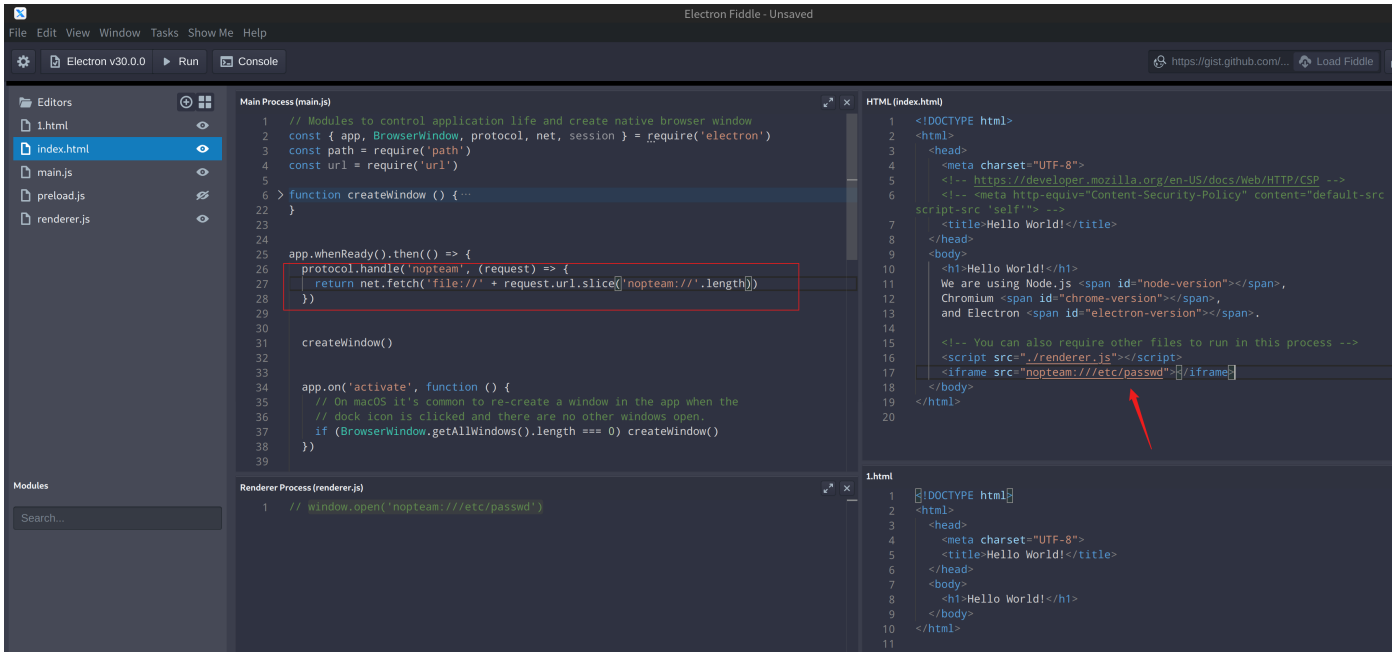
```
const { app, protocol, net } = require('electron')

app.whenReady().then(() => {
  protocol.handle('nopteam,', (request) =>
    net.fetch('file://' + request.url.slice('nopteam://'.length)))
})
```

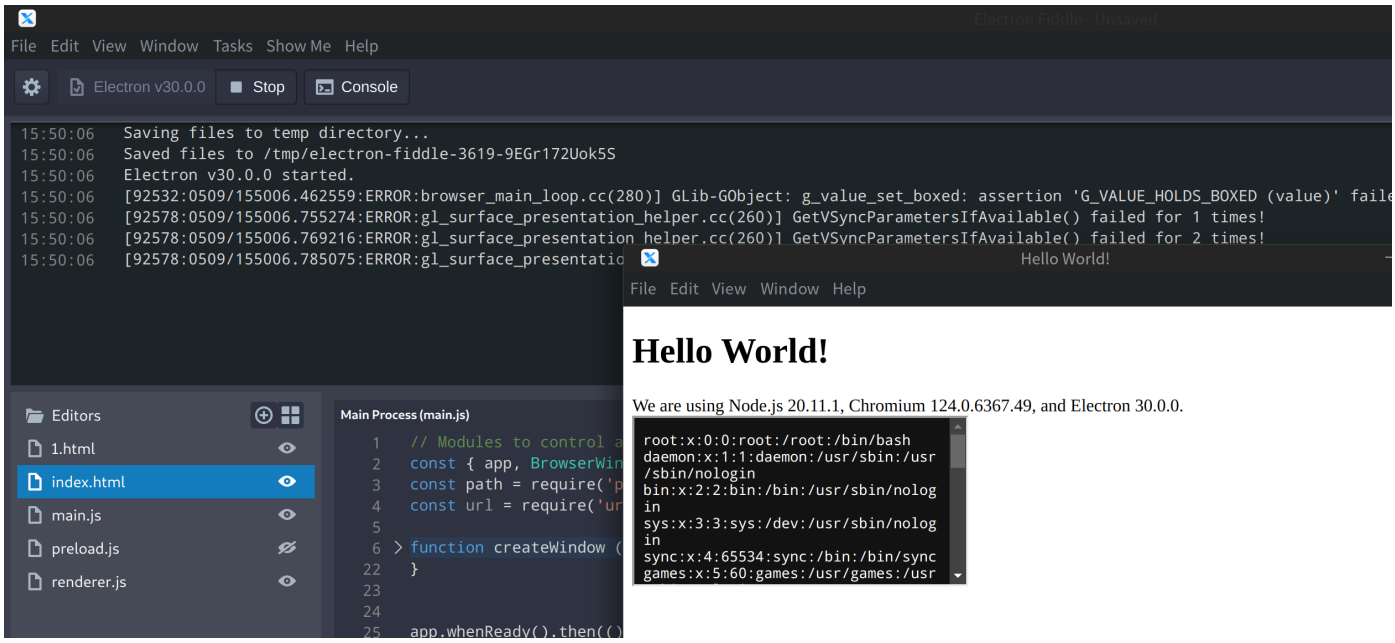
在渲染页面的 `JavaScript` 中使用 `nopteam://` 协议



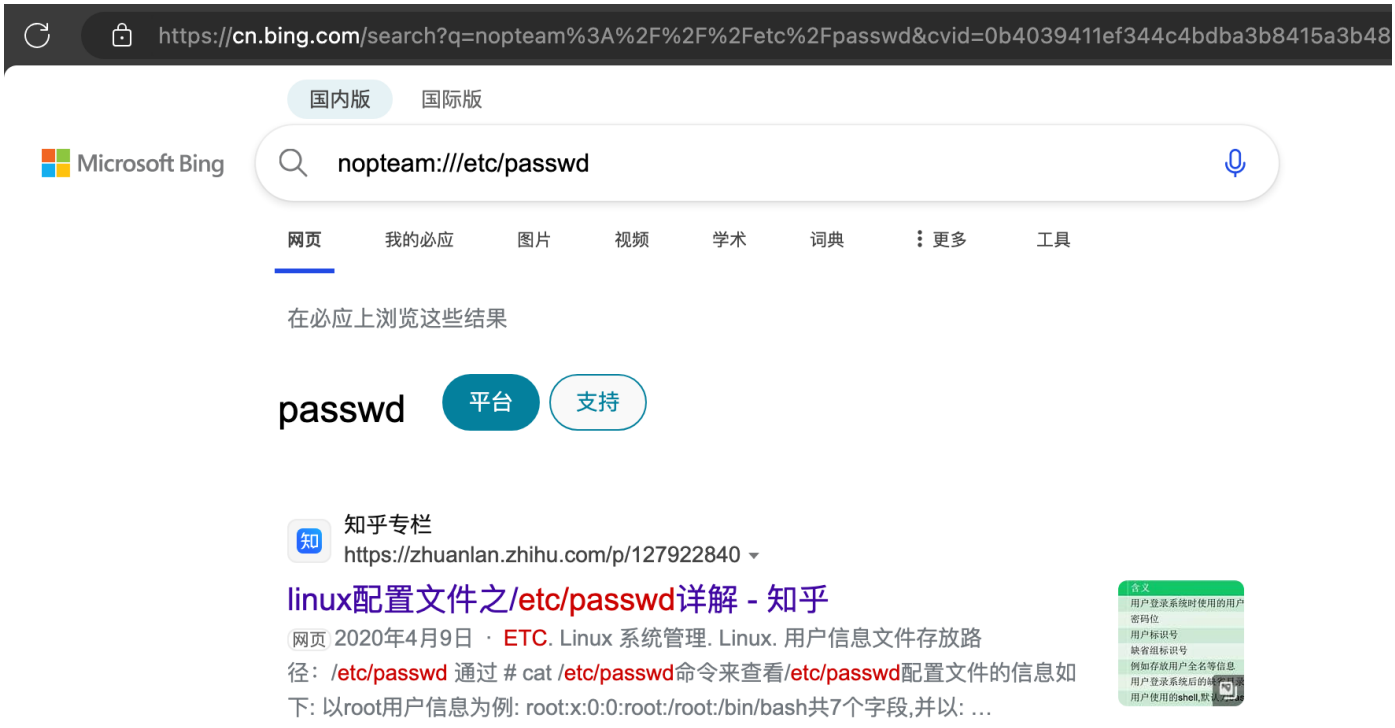


在 HTML 标签内使用 `nopteam://` 协议





但只限于程序内容，在浏览器中输入 `nopteam:///etc/passwd` 并不可以打开我们的程序



2. 注册协议到特定 session

如果我们想将自定义的协议注册到特定的 `session` ，而不是默认的，可以使用以下代码

```
const { app, BrowserWindow, net, protocol, session } = require('electron')
const path = require('node:path')
const url = require('url')

app.whenReady().then(() => {
  const partition = 'persist:example'
  const ses = session.fromPartition(partition)

  ses.protocol.handle('atom', (request) => {
    const filePath = request.url.slice('atom://'.length)
    return net.fetch(url.pathToFileURL(path.join(__dirname,
filePath)).toString())
  })

  const mainWindow = new BrowserWindow({ webPreferences: { partition } })
})
```

这里涉及两个概念， `session` 和 `partition`

Partition: 分区（`Partition`）是一种机制，用于将不同部分的应用数据隔离开来。每个分区都有其独立的Cookies、本地存储（localStorage）、IndexedDB等数据存储空间。

当你创建一个新的BrowserWindow或者WebContents时，可以通过指定`partition`参数来决定这个新窗口或页面的数据是否与其他窗口共享，或者是否持久化存储。

- 当你设置`partition:'persist:name'`时，Electron 会为该窗口创建一个持久化的分区，即使应用重启，这个分区中的数据（如Cookie）也会被保留。
- 如果不指定或者使用`partition:''`（空字符串），则使用一个临时的、匿名的分区，关闭窗口后相关数据会被清除

Session: 会话（Session）在 Electron 中是一个更高级的概念，它代表了一组配置和行为，用于控制网络请求、缓存策略、Cookie管理等。一个Session可以有自己的存储、Cookie和其他设置，并且可以被多个WebContents共享。

- **创建Session**: 你可以通过`session.fromPartition()`方法创建一个基于特定分区名的Session实例，或者直接使用`session.defaultSession`来获取应用的默认Session。
- **控制行为**: Session允许你控制例如是否允许使用缓存、是否发送Referer头、代理设置等网络行为，以及管理权限、证书等安全相关的方面。

这样上述代码就比较好理解了

3. protocol 模块的方法

1) registerSchemesAsPrivileged

```
protocol.registerSchemesAsPrivileged(customSchemes)
```

注意. 此方法只能在`app`的`ready`事件触发前调用，且只能调用一次

此方法用来对我们自定义协议(`scheme`)进行配置，可以注册为一个标准、安全、允许注册ServiceWorker、支持获取API、流视频/音频和V8代码缓存的协议

示例代码如下

```
const { protocol } = require('electron')
protocol.registerSchemesAsPrivileged([
  { scheme: 'foo', privileges: { bypassCSP: true } }
])
```

`CustomScheme` 对象的内容结构如下

- `scheme` 字符串 - 自定义的计划，可以被按选项注册。
- `privileges` Object (可选)
  - `standard` boolean (可选) 默认为false

    是否注册为标准协议
  - `secure` boolean (可选) - 默认为false

    是否被视为安全协议，意味着它可以请求HTTPS资源而不会触发混合内容警告，并且在Web内容中可能不受同源策略的某些限制
  - `bypassCSP` boolean (可选) - 默认为false

    如果设为 `true`，则该协议下的资源可以绕过页面的Content Security Policy (CSP)策略限制，这在某些特定场景下可能有用，但也可能带来安全风险
  - `allowServiceWorkers` boolean (可选) - 默认为false

    允许在该协议下注册和使用Service Workers
  - `supportFetchAPI` boolean (可选) - 默认为false

    启用后，允许在该协议下通过 `fetch` API进行网络请求，这对于现代Web应用中异步数据获取非常重要
  - `corsEnabled` boolean (可选) - 默认为false

    启用跨源资源共享(CORS)，允许该协议下的资源被其他源的Web页面请求，这对于跨域数据交换是必需的
  - `stream` boolean (可选) - 默认为 false

    如果设为 `true`，则支持通过流(Streams)来传输数据，这在处理大文件或连续数据时可以提高效率和响应性
  - `codeCache` boolean (可选) - 默认为 false

    启用支持 v8 代码缓存，只有在 `standard` 被设置为 true 时有效

标准scheme遵循 RFC 3986 所设定的 URI泛型语法 。 例如, `http` 和 `https` 是标准协议, 而 `file` 不是

按标准将一个scheme注册, 将保证相对和绝对资源在使用时能够得到正确的解析。 否则, 该协议将表现为 `file` 协议, 而且，这种文件协议将不能解析相对路径

例如, 当您使用自定义协议加载以下内容时，如果你不将其注册为标准scheme, 图片将不会被加载, 因为非标准scheme无法识别相对 路径:

```
<body>
  <img src='test.png'>
</body>
```

注册一个scheme作为标准scheme将允许其通过FileSystem 接口访问文件。 否则, 渲染器将会因为该scheme，而抛出一个安全性错误。

在非标准 schemes 下，网络存储 Api (localStorage, sessionStorage, webSQL, indexedDB, cookies) 默认是被禁用的。 所以一般来说如果你想注册一个自定义协议来替换 `http` 协议，你必须将其注册为标准 scheme：

如果 Protocols 需要使用流 (http 和 stream 协议) 应设置 `stream: true`。 `<video>` 和 `<audio>` HTML 元素默认需要协议缓冲其响应内容。 `stream` 标志将这些元素配置为正确的流媒体响应

2) handle

这个方法用来注册协议，并关联协议处理程序

```
protocol.handle(scheme, handler)
```

- `scheme` 协议名，例如 `https` 不包含
- `handler` 协议处理程序，是一个协议处理函数

当Electron遇到匹配到 `scheme` 的URL请求时 `handler` 会被调用。这个函数接收一个 `request` 对象作为参数，并且通常需要调用一个回调函数，返回值是一个 `Promise<GlobalResponse>`

> request 对象具体结构参考

> https://nodejs.org/api/globals.html#request
>
> https://developer.mozilla.org/en-US/docs/Web/API/Request
>
> response 对象具体结构参考
>
> https://developer.mozilla.org/en-US/docs/Web/API/Response

官方案例如下

```
const { app, net, protocol } = require('electron')
const path = require('node:path')
const { pathToFileURL } = require('url')

protocol.registerSchemesAsPrivileged([
  {
    scheme: 'app',
    privileges: {
      standard: true,
      secure: true,
      supportFetchAPI: true
    }
  }
])

app.whenReady().then(() => {
  protocol.handle('app', (req) => {
    const { host, pathname } = new URL(req.url)
    if (host === 'bundle') {
      if (pathname === '/') {
        return new Response('<h1>hello, world</h1>', {
          headers: { 'content-type': 'text/html' }
        })
      }
      // NB, this checks for paths that escape the bundle, e.g.
      // app://bundle/../../secret_file.txt
      const pathToServe = path.resolve(__dirname, pathname)
      const relativePath = path.relative(__dirname, pathToServe)
```

```
      const isSafe = relativePath && !relativePath.startsWith('..') &&
!path.isAbsolute(relativePath)
      if (!isSafe) {
        return new Response('bad', {
          status: 400,
          headers: { 'content-type': 'text/html' }
        })
      }

      return net.fetch(pathToFileURL(pathToServe).toString())
    } else if (host === 'api') {
      return net.fetch('https://api.my-server.com/' + pathname, {
        method: req.method,
        headers: req.headers,
        body: req.body
      })
    }
  })
})
```

3) unhandle

```
protocol.unhandle(scheme)
```

这个就很好理解了，取消注册协议

4) isProtocolHandled

```
protocol.isProtocolHandled(scheme)
```

一个 `scheme` 是否被注册为了一个协议，就是看一个协议有没有被注册过

> 参考文章

> https://www.electronjs.org/zh/docs/latest/api/protocol

0x03 全局注册自定义协议

程序内部协议只能在程序内部使用，如果我们注册一个 `nopteam` 协议，希望在浏览器里输入 `nopteam://index?id=1` 时不仅可以唤醒我们的应用，应用还可以获取到链接内容，并且根据实际内容进行对应处理

1. 效果展示

我们希望 `id=1` 的时候，主窗口渲染 `1.html` ， `id=2` 时，主窗口渲染 `2.html`

`1.html`

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
  <meta http-equiv="Content-Security-Policy" content="default-src 'self';
script-src 'self'">
  <meta http-equiv="X-Content-Security-Policy" content="default-src 'self';
script-src 'self'">
  <title>1.html</title>
</head>

<body>
  <h1>I am 1.html !</h1>
</body>

</html>
```

`2.html`

```html
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
  <meta http-equiv="Content-Security-Policy" content="default-src 'self';
script-src 'self'">
  <meta http-equiv="X-Content-Security-Policy" content="default-src 'self';
script-src 'self'">
  <title>2.html</title>
</head>

<body>
  <h1>I am 2.html !</h1>
</body>

</html>
```

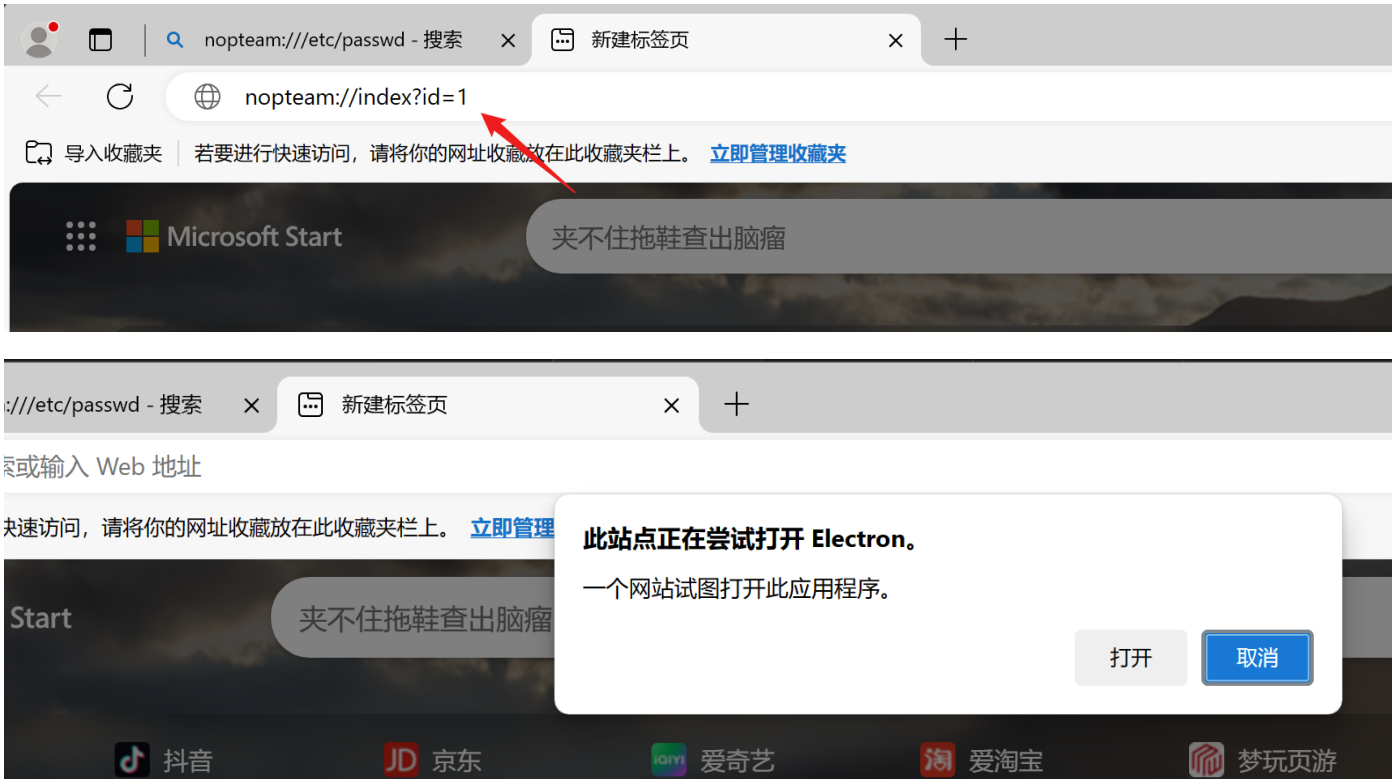
`main.js`

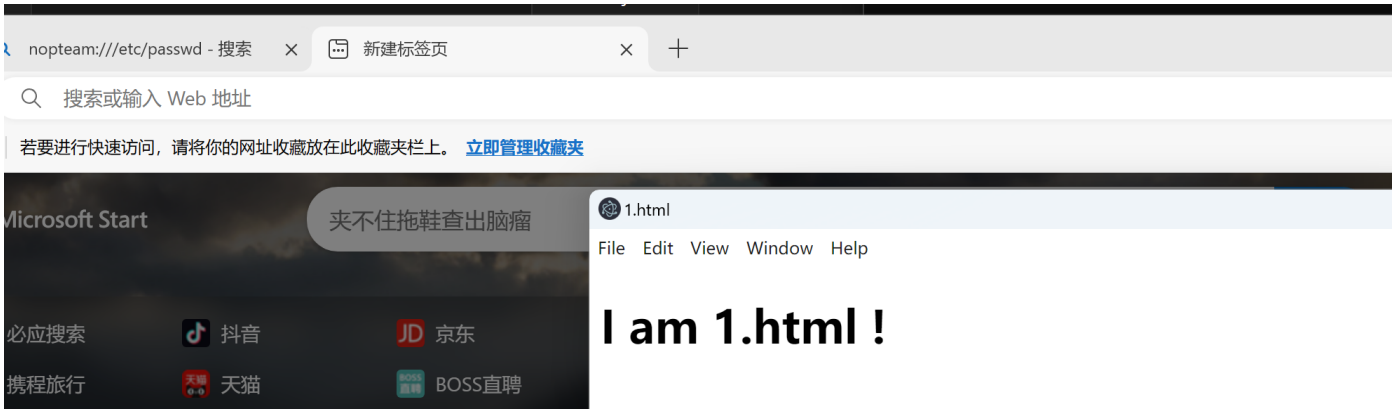
```js
// Modules to control application life and create native browser window
const { app, BrowserWindow, ipcMain, shell, dialog } =
require('electron/main')
const path = require('node:path')
const url = require('url')

let mainWindow

if (process.defaultApp) {
  if (process.argv.length >= 2) {
    app.setAsDefaultProtocolClient('nopteam', process.execPath,
[path.resolve(process.argv[1])])
  }
} else {
  app.setAsDefaultProtocolClient('nopteam')
}
```

```
const gotTheLock = app.requestSingleInstanceLock()

if (!gotTheLock) {
  app.quit()
} else {
  app.on('second-instance', (event, commandLine, workingDirectory) => {
    // Someone tried to run a second instance, we should focus our window.
    if (mainWindow) {
      if (mainWindow.isMinimized()) mainWindow.restore()
      mainWindow.focus()
    }

    const parsedUrl = new URL(commandLine.pop())
    const page_id = parsedUrl.searchParams.get('id')

    let page_path
    if (page_id === '1') {
      page_path = '1.html'
    } else if (page_id === '2') {
      page_path = '2.html'
    } else {
      app.quit()
    }

    console.log(page_path)
    mainWindow.loadFile(path.join(__dirname, page_path))
  })

  // Create mainWindow, load the rest of the app, etc...
  app.whenReady().then(() => {
    createWindow()
  })

  app.on('open-url', (event, url) => {
    dialog.showErrorBox('Welcome Back', `You arrived from: ${url}`)
  })
}

function createWindow () {
```

```
  // Create the browser window.
  mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })

  mainWindow.loadFile(path.join(__dirname, 'index.html'))
}

app.on('window-all-closed', function () {
  if (process.platform !== 'darwin') app.quit()
})
```

运行一次后，就会注册全局协议 `nopteam` ，之后在浏览器里输入 `nopteam://index?id=1`





当输入 `nopteam://index?id=2` 时

成功解析了我们的自定义 url

注册全局协议，主要使用 `app` 模块的一些方法

2. app.setAsDefaultProtocolClient

将当前可执行文件的设置为协议(也就是 URI scheme) 的默认处理程序。 该方法允许你将应用更深入地集成到操作系统中

```
app.setAsDefaultProtocolClient(protocol[, path, args])
```

- protocol 协议名称，字符串类型
- path 可选项， `Electron` 执行路径，默认为 `process.execPath` ，仅在 Windows 平台有用
- args 可选项，传递给可执行文件的参数，默认是一个空数组，仅在 Windows 平台有用

注意: 在 macOS 上，您只能注册已添加到应用程序的 `info.plist` 中的协议，这个列表在运行时不能修改。 然而，你可以在构建时通过 Electron Forge, Electron Packager, 或通过文本编辑器编辑 `info.plist` 文件的方式修改

3. app.removeAsDefaultProtocolClient

此方法检查当前可执行程序是否是协议(也就是URI scheme) 的默认处理程序。 如果是，则会将应用移除默认处理器

```
app.removeAsDefaultProtocolClient(protocol[, path, args])
```

4. app.isDefaultProtocolClient

当前可执行程序是否是协议(也就是URI scheme) 的默认处理程序

```
app.isDefaultProtocolClient(protocol[, path, args])
```

5. app.getApplicationNameForProtocol

此方法返回URL协议(也就是URI scheme) 的默认处理器的应用程序名称

```
app.getApplicationNameForProtocol(url)
```

- url 要检查的协议名称的 URL，不同于家族中的其他方法，该方法接收至少包含 `://` (例如： `https://`)的完整URL

```
  }

  const parsedUrl = new URL(commandLine.pop())
  const page_id = parsedUrl.searchParams.get('id')

  let page_path
  if (page_id === '1') {
    page_path = '1.html'
  } else if (page_id === '2') {
    page_path = '2.html'
  } else {
    app.quit()
  }

  console.log(page_path)
  mainWindow.loadFile(path.join(__dirname, page_path))
})

// Create mainWindow, load the rest of the app, etc...
app.whenReady().then(() => {
  createWindow()
  console.log(app.getApplicationNameForProtocol("nopteam://index?id=1"))
})
app.on('open-url', (event, url) => {
  dialog.showErrorBox('Welcome Back', `You arrived from: ${url}`)
})
}
```

```
PS C:\Users\join\Desktop\forge_test\my-app> npm run start

> my-app@1.0.0 start
> electron-forge start

✔Checking your system
✔Locating application
✔Loading configuration
✔Preparing native dependencies [2s]
✔Running generateAssets hook


Electron
```

index.html

File Edit View Window Help

Hello World!

不同平台值可能不完全相同

6. app.getApplicationInfoForProtocol

此方法返回包含应用程序名称，图标和默认协议处理器路径(也就是URI scheme) 的Promise

```
app.getApplicationInfoForProtocol(url)
```

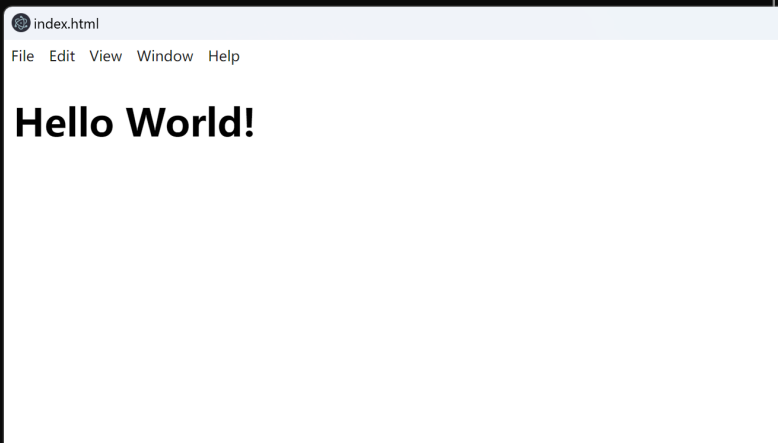
- `url` string - 要检查的协议名称的 URL。 不同于家族中的其他方法，该方法接收至少包含 `://` (例如： `https://`)的完整URL

返回 `Promise<Object>` - resolve 包含以下内容的 object：

- `icon` NativeImage - 处理协议的应用程序的显示图标。
- `path` string - 处理协议的应用程序的安装路径。
- `name` string - 处理协议的应用程序的显示名称。

```
  const parsedUrl = new URL(commandLine.pop())
  const page_id = parsedUrl.searchParams.get('id')

  let page_path
  if (page_id === '1') {
    page_path = '1.html'
  } else if (page_id === '2') {
    page_path = '2.html'
  } else {
    app.quit()
  }

  console.log(page_path)
  mainWindow.loadFile(path.join(__dirname, page_path))
})

// Create mainWindow, load the rest of the app, etc...
app.whenReady().then(() => {
  createWindow()
  app.getApplicationInfoForProtocol("nopteam://index?id=1").then((o) => {
    console.log(o)
  })

})
```

```
PS C:\Users\join\Desktop\forge_test\my-app> npm start

> my-app@1.0.0 start
> electron-forge start

✔Checking your system
✔Locating application
✔Loading configuration
✔Preparing native dependencies [1s]
✔Running generateAssets hook


{
  path: 'C:\\Users\\join\\Desktop\\forge_test\\my-app\\node_modules\\electron\\dist\\electron.exe',
  name: 'Electron',
  icon: NativeImage {
    toPNG: [Function: toPNG],
    toJPEG: [Function: toJPEG],
    toBitmap: [Function: toBitmap],
    getBitmap: [Function: getBitmap],
    getScaleFactors: [Function: getScaleFactors],
    getNativeHandle: [Function: getNativeHandle],
    toDataURL: [Function: toDataURL],
    isEmpty: [Function: isEmpty],
    getSize: [Function: getSize],
    setTemplateImage: [Function: setTemplateImage],
    isTemplateImage: [Function: isTemplateImage],
    isMacTemplateImage: [Getter/Setter],
    resize: [Function: resize],
    crop: [Function: crop],
    getAspectRatio: [Function: getAspectRatio],
    addRepresentation: [Function: addRepresentation]
  }
}
```



> 参考文章
>
> https://www.electronjs.org/zh/docs/latest/tutorial/launch-app-from-url-in-another-app

> https://www.electronjs.org/docs/latest/api/app#appsetasdefaultprotocolclientprotocol-path-args

0x04 漏洞案例

这种注册自定义协议具体实现方法不同程序不一致，所以在做安全检查时，也需要根据实际情况，接下来列举几个曾经在注册自定义协议方面出现的问题

需要注意的是，外部引用的安全防护代码可能不会针对自定义协议进行防护，这也是造成很多漏洞的直接原因

`CVE-2018-1000006`

这个漏洞是个Windows 平台独有的漏洞，在注册全局协议时，用户可以控制 URL，打开特定的 URL 时，URL中的一部分可能会闭合处理程序的语法，导致另一部分成为传递给处理程序的参数，配合 `Chromium` 的一些特殊参数，最终导致命令执行，下方参考链接中先知社区的文章对其分析得比较好，建议观看

> 参考文章
>
> https://www.electronjs.org/blog/protocol-handler-fix
>
> https://xz.aliyun.com/t/1994?time__1311=n4%2Bxni0QDQdYqDvPBKDsL3ObDcBIKKriTo4D&alichlgref=https%3A%2F%2Fwww.google.com%2F
>
> https://blog.doyensec.com/2018/05/24/electron-win-protocol-handler-bug-bypass.html

`typora (CVE-2023-2317)`

> https://xz.aliyun.com/t/12822?time__1311=mqmhq%2BxfxIhGkDlxGo%2Bzd4Dv5TNDjETD&alichlgref=https%3A%2F%2Fwww.google.com%2F

低于1.67版本的Typora存在代码执行漏洞，通过在标签中加载 `typora://app/typemark/updater/update.html` 实现在Typora主窗口的上下文中运行任意JavaScript代码

0x05 总结

注册自定义协议通常用来实现特殊功能，比如深度集成应用程序与特定的网络服务、提升用户体验或实现安全的数据交换、插件等

自定义协议关联的处理程序几乎没有特别多的共性，完全由需求决定，因此可能会由于不够健硕的代码而带来一些安全风险，这部分漏洞的挖掘需要对 `protocol` 和 `app` 模块的相关方法进行分析，查找攻击的可能